Programming Languages and Techniques (CIS1200)

Lecture 19

GUI library: Events and State

Chapter 18

Looking Ahead…

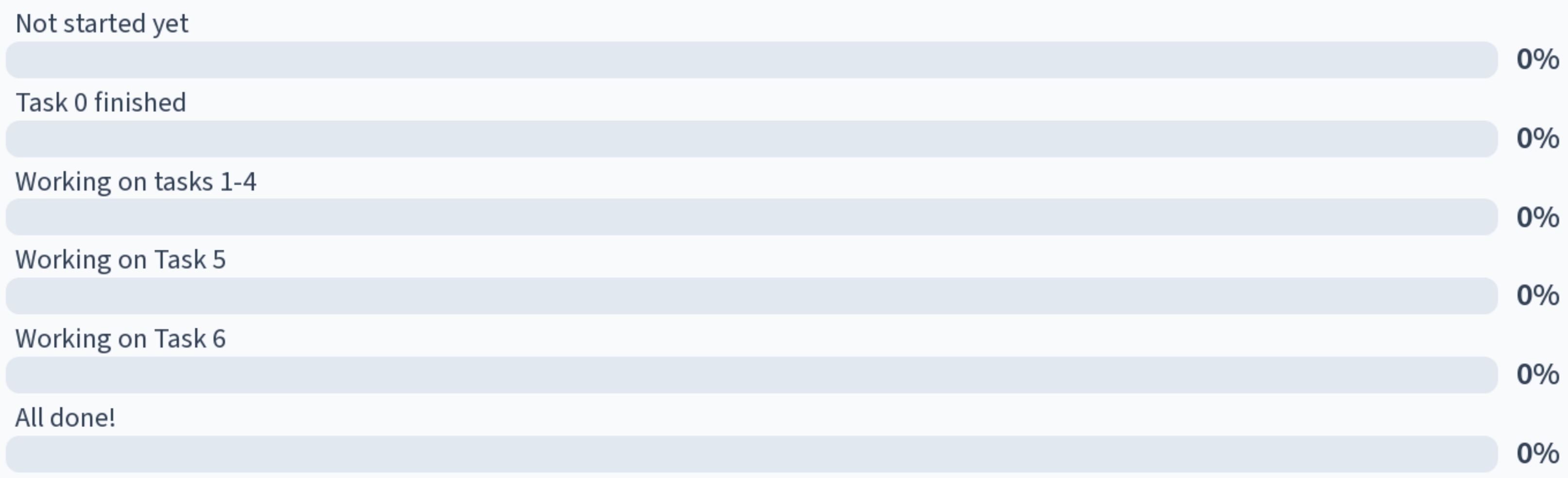
- HW05: GUI Programming
  - due Tuesday **March 18** after Spring Break
  - ***START NOW!!***
  - aim to complete by this Friday

- Friday March 7th: NO CLASS
- No classes/recitations/TA office hours during Spring Break!

- Two weeks after break will move quickly
  - Transition to Java: Monday, March 17
  - Java Bootcamp: Wednesday, March 19
  - Homework 06 (Java) due: Tuesday, March 25
  - Midterm 2: Friday, March 28
    - OCaml: ASM, mutability, queues/deques, closures, GUI, and Java basics

20: How far along are you in HW05: GUI Programming?

0



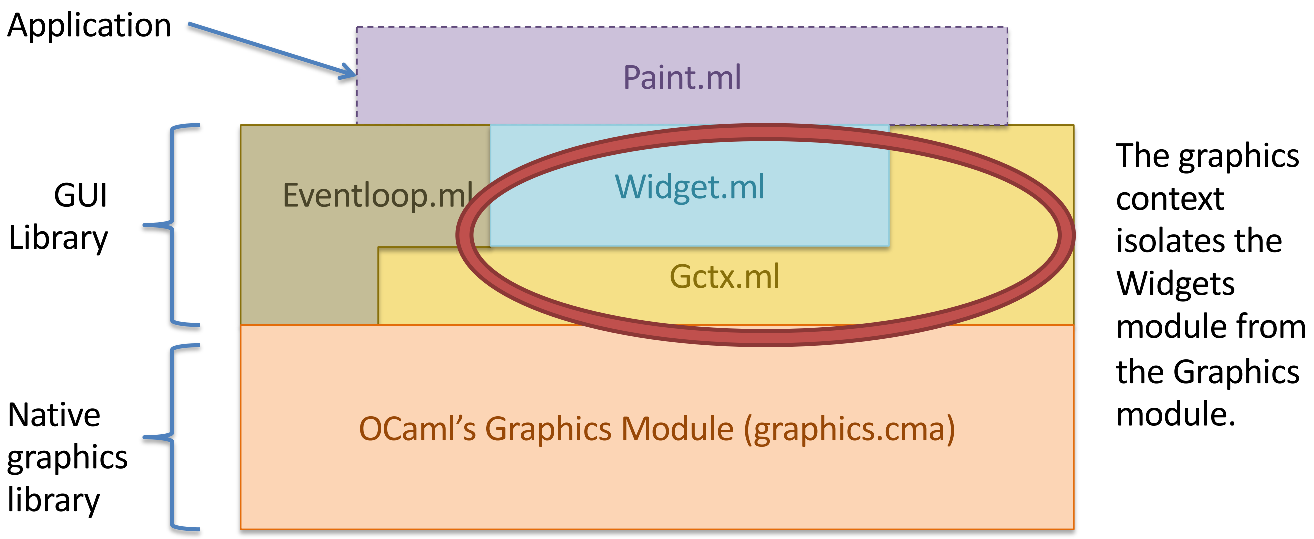
Review: Widget Layout

Building blocks of GUI applications

see simpleWidget.ml in GUI Demo Code project

Widget Layout

- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to the widget’s current position


Application
Paint.ml
GUI Library
Eventloop.ml
Widget.ml
Gctx.ml
Native graphics library
OCaml’s Graphics Module (graphics.cma)
The graphics context isolates the Widgets module from the Graphics module.

Layout with Simple Widgets

simpleWidget.mli

```
(* An interface for simple GUI widgets *)
type widget = {
   repaint : Gctx.gctx -> unit;
   size    : unit -> (int * int)
}
val label  : string -> widget
val space  : int * int -> widget
val border : widget -> widget
val hpair  : widget -> widget -> widget
val canvas : int * int -> (Gctx.gctx -> unit) -> widget
```

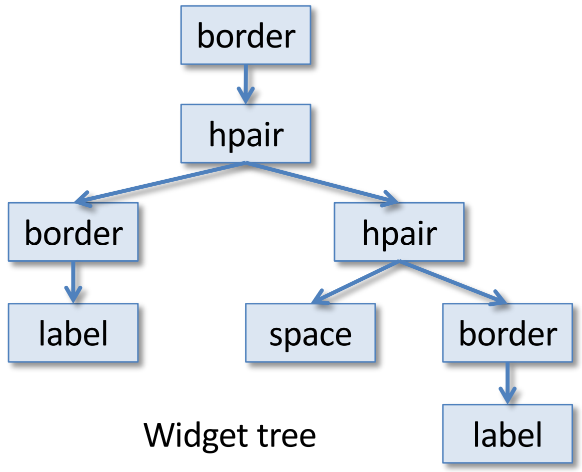
- You can ask a simple widget to repaint itself
- You can ask a simple widget to tell you its size
- (We'll talk about handling events later)

- Repainting is relative to a graphics context

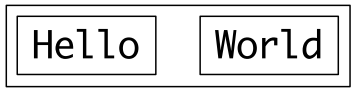
Widget Hierarchy Pictorially

swdemo.ml

```
(* Create some simple label widgets *)
let l1 = label "Hello"
let l2 = label "World"
(* Compose them horizontally, adding some borders *)
let h = border (hpair (border l1)
                      (hpair (space (10,10)) (border l2)))
```

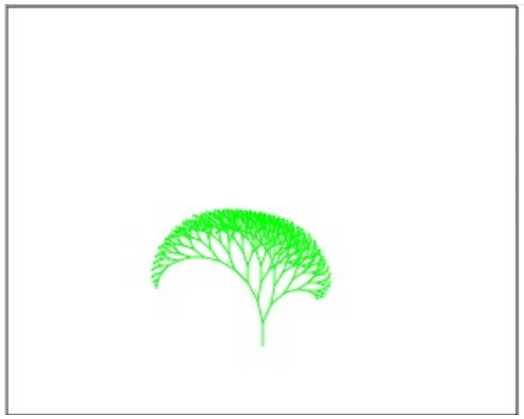


Widget tree



On the screen

"Fractal Tree" application



fractalTree.ml

```
(* Use the graphics context to draw a fractal tree *)
let paint_tree (g:Gctx.gctx) : unit = …

(* Create a canvas widget that draws the fractal tree *)
let c = border (canvas (300, 240) paint_tree)
```

Widget Implementations

```
(* A simple widget that puts some text on the screen *)
let label (s:string) : widget =
{
  repaint = (fun (g:Gctx.gctx) -> Gctx.draw_string g (0,0) s);
  size = (fun () -> Gctx.text_size s)
}
```

simpleWidget.ml

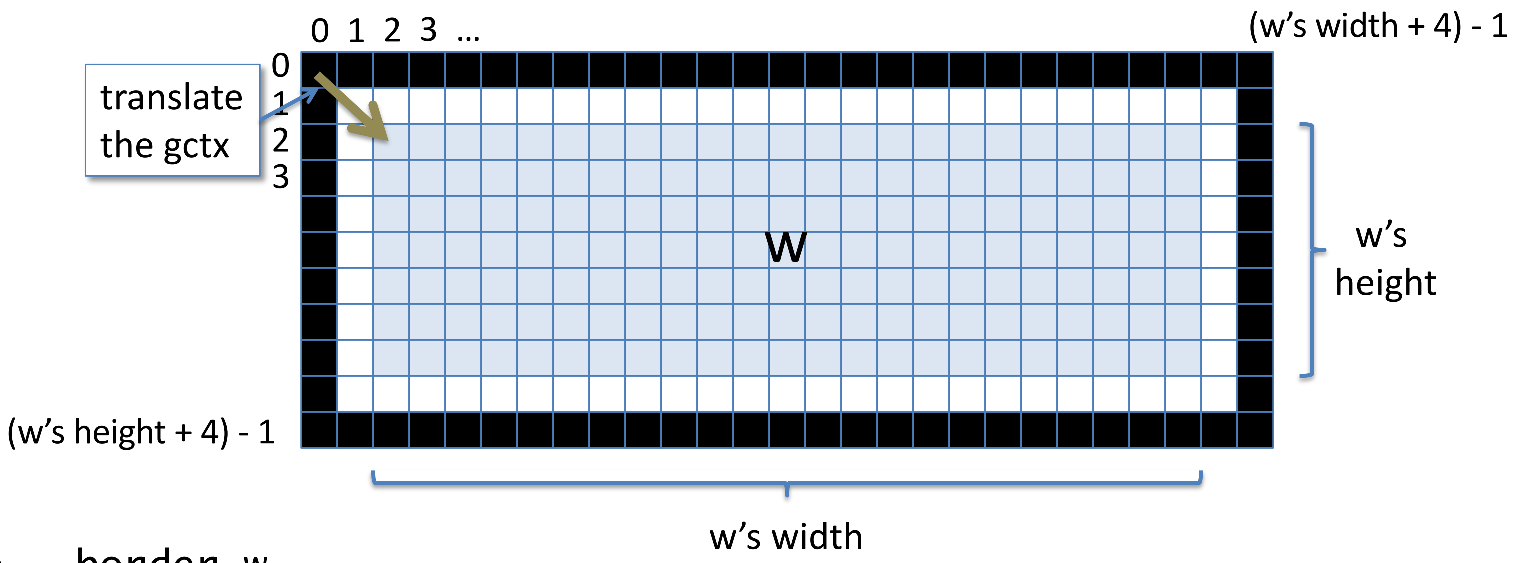
```
(* A "blank" area widget -- it just takes up space *)
let space ((w,h):int*int) : widget =
{
  repaint = (fun (_:Gctx.gctx) -> ());
  size = (fun () -> (w,h))
}
```

simpleWidget.ml

Nested Widgets

Containers and Composition

The Border Widget Container



```
let b = border w
```

- Draws a one-pixel-wide border (+ a one-pixel space) around contained widget w
- b's size is slightly larger than w's (+4 pixels in each dimension)
- b's repaint method must call w's repaint method
- When b asks w to repaint, b must *translate* the gctx to (2,2) to account for the displacement of w from b's origin

The Border Widget

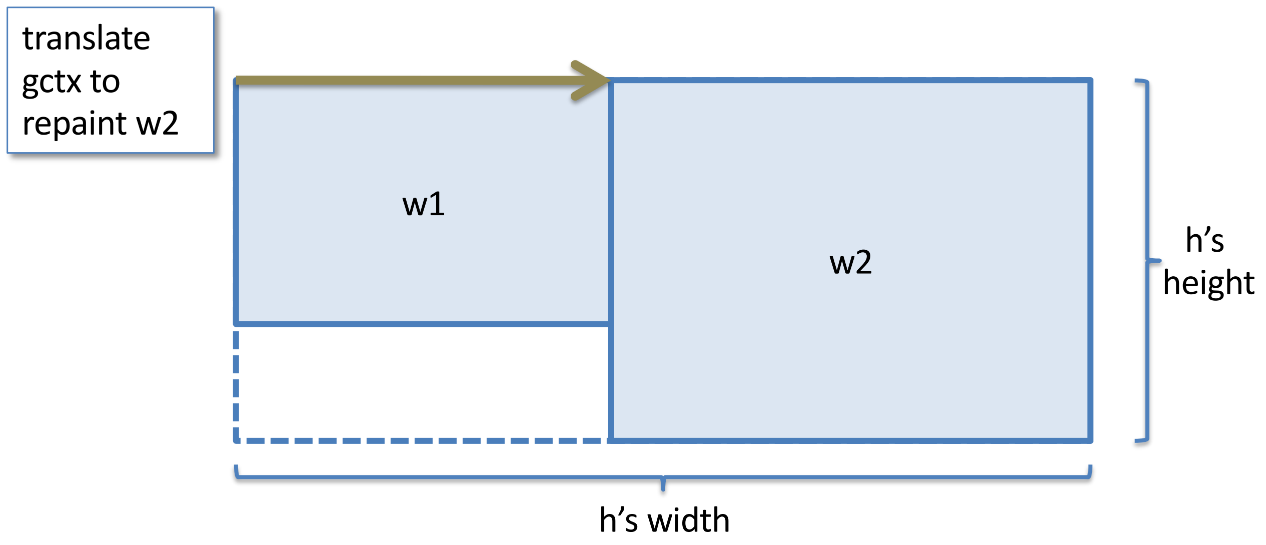
simpleWidget.ml

```
let border (w:widget):widget =
{
repaint = (fun (g:Gctx.gctx) ->
  let (width,height) = w.size () in
  let x = width + 3 in
  let y = height + 3 in
  Gctx.draw_line g (0,0) (x,0);
  Gctx.draw_line g (0,0) (0,y);
  Gctx.draw_line g (x,0) (x,y);
  Gctx.draw_line g (0,y) (x,y);
  let gw = Gctx.translate g (2,2) in
  w.repaint gw);

size = (fun () ->
  let (width,height) = w.size () in
  (width+4, height+4))
}
```

Draw the border

Display the interior

The hpair Widget Container



- `let h = hpair w1 w2`
- Creates a horizontally adjacent pair of widgets
- Aligns them by their top edges
- Size is the *sum* of their widths and *max* of their heights

The hpair Widget

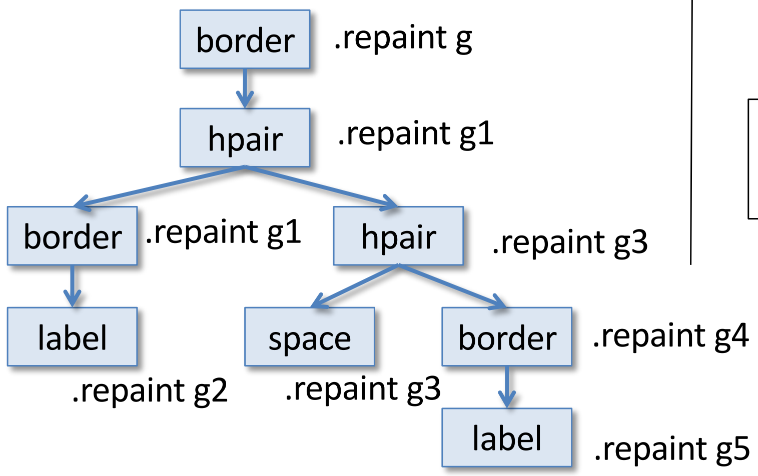
simpleWidget.ml

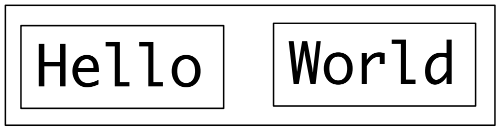
```
let hpair (w1: widget) (w2: widget) : widget =
  {
    repaint = (fun (g: Gctx.gctx) ->
            let (x1, _) = w1.size () in begin
              w1.repaint g;
              w2.repaint (Gctx.translate g (x1,0))
              (* Note translation of the Gctx *)
            end);

    size = (fun () ->
            let (x1, y1) = w1.size () in
            let (x2, y2) = w2.size () in
            (x1 + x2, max y1 y2))
  }
```

Translate the Gctx to shift w2's position relative to widget-local origin.

Drawing: Containers

Container widgets propagate repaint commands to their children, with appropriately modified graphics contexts:





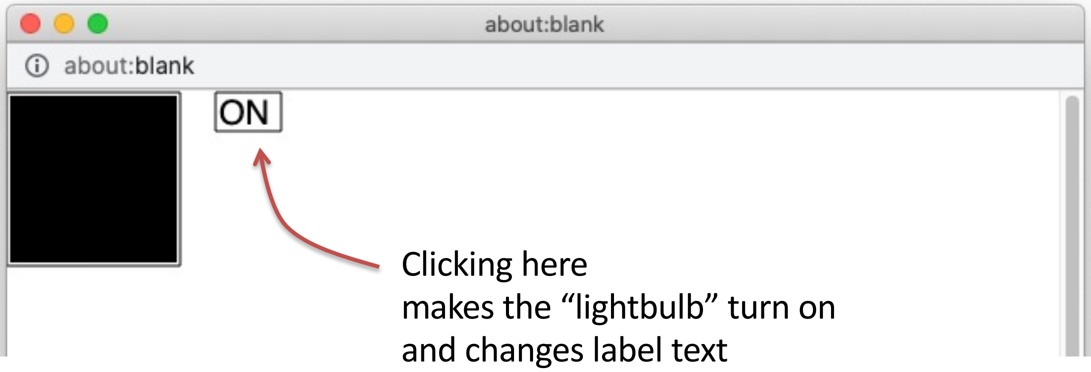
```
let l1 = label "Hello"
let l2 = label "World"
let h = border (hpair (border l1)
                 (hpair (space (10,10))
                   (border l2)))
;; h.repaint Gctx.top_level
```

```
g1 = Gctx.translate g (2,2)
g2 = Gctx.translate g1 (2,2)
g3 = Gctx.translate g1 (hello_width,0)
g4 = Gctx.translate g3 (space_width,0)
g5 = Gctx.translate g4 (2,2)
```

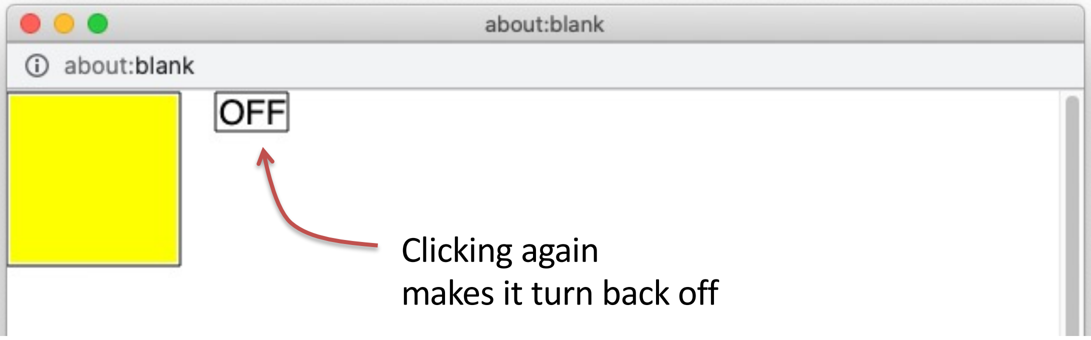
Coding with Simple Widgets

see swdemo.ml

"lightbulb" demo

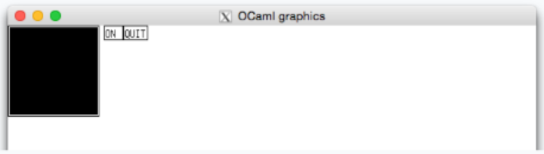


Clicking here
makes the “lightbulb” turn on
and changes label text



Clicking again
makes it turn back off

19: Do you know how you would use the (simple) widget library to define the layout of this application?



```
type widget = {
   repaint : Gctx.gctx -> unit;
   size    : unit -> (int * int)
}
val label  : string -> widget
val space  : int * int -> widget
val border : widget -> widget
val hpair  : widget -> widget -> widget
val canvas : int * int -> (Gctx.gctx -> unit) -> widget
```

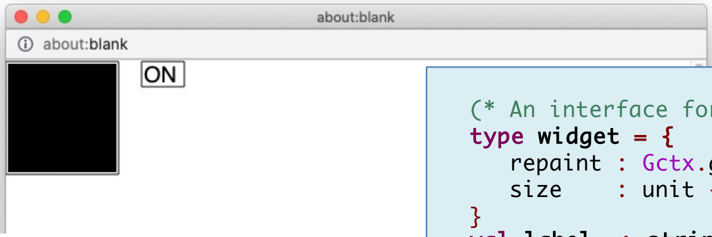
I don't know how to start 0%

I may have it, but I'm not sure 0%

I'm sure I've got it 0%

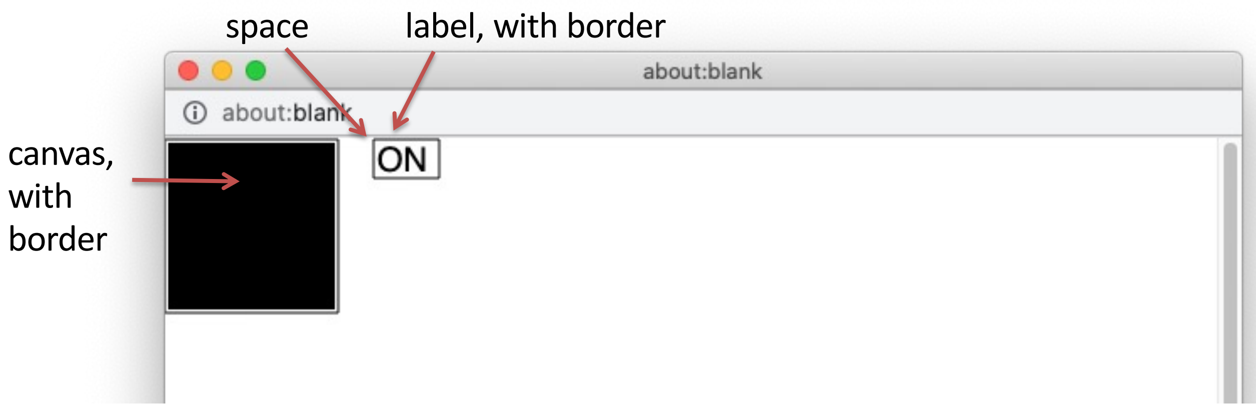
Do you know how you would use the (simple) widget library to define the layout of this lightbulb application?

1. I'm not sure how to start.
2. I may have it, but I'm not sure.
3. Sure! No problem.



```
(* An interface for simple GUI widgets *)
type widget = {
   repaint : Gctx.gctx -> unit;
   size    : unit -> (int * int)
}
val label  : string -> widget
val space  : int * int -> widget
val border : widget -> widget
val hpair  : widget -> widget -> widget
val canvas : int * int -> (Gctx.gctx -> unit) -> widget
```

"lightbulb" demo layout

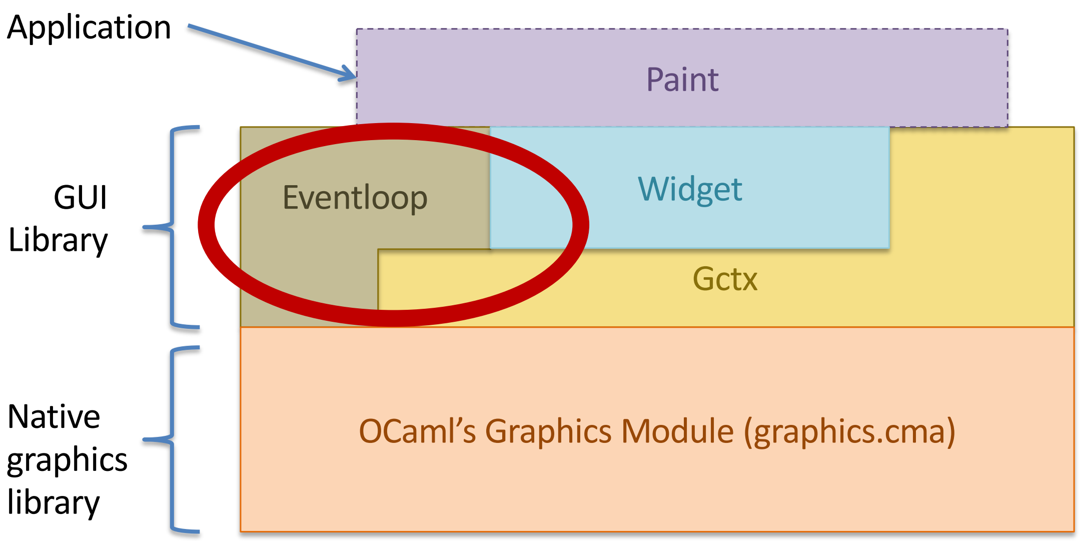


swdemo.ml

```
let onoff = border (label "ON")

let paint_bulb (g: Gctx.gctx) : unit = …

let bulb = border (canvas (100, 100) paint_bulb)

let top : widget = hpair bulb (hpair (space (20, 20)) onoff)
```

Events and Event Handling

Project Architecture



Event loop with event handling

```
let run (w:widget) : unit =
  let g = Gctx.top_level  in      …create the initial gctx…
  w.repaint g;                    …display the widget
  Graphics.loop                   …wait for user input
    (fun e ->
      clear_graph ();
      w.handle g e;               …inform widget about the event…
      w.repaint g)                …update the widget's appearance…
```

Eventloop

```
let rec loop (f: event -> unit) : unit =
  let e = wait_next_event () in                … wait for OS event
  f e;                                         … call function argument
  loop f                                       … tail recursion
```

Graphics

Events

gctx.mli

```
type event

val wait_for_event : unit -> event

type event_type =
  | KeyPress of char  (* User pressed a key                  *)
  | MouseDown         (* Mouse Button pressed, no movement  *)
  | MouseUp           (* Mouse button released, no movement *)
  | MouseMove         (* Mouse moved with button up          *)
  | MouseDrag         (* Mouse moved with button down        *)

val event_type : event -> event_type
val event_pos  : event -> gctx -> position
```

Remember:
The graphics context translates the location of the event *to widget-local coordinates*

Reactive Widgets

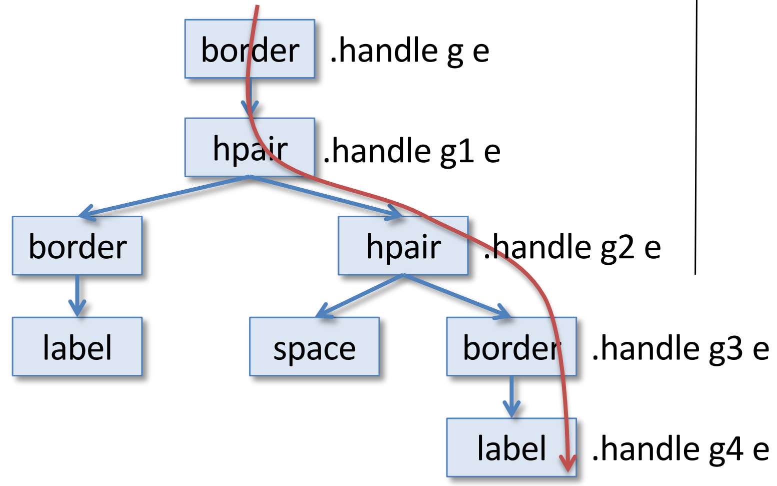
widget.mli

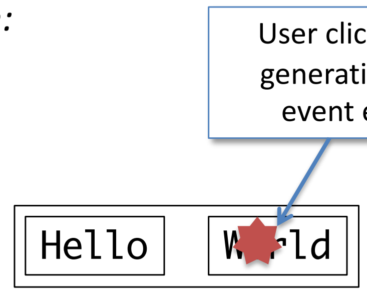
```
type widget = {
  repaint : Gctx.gctx -> unit;
  size    : unit -> Gctx.dimension;
  handle  : Gctx.gctx -> Gctx.event -> unit
}
```

- Widgets now have a “method” for handling events
  - The eventloop waits for an event and then gives it to the root widget
  - The widgets forward the event down the tree, according to the position of the event

Event-handling: Containers

Container widgets propagate events to their children:





Widget tree

```
g1 = Gctx.translate g (2,2)
g2 = Gctx.translate g1 (hello_width,0)
g3 = Gctx.translate g2 (space_width,0)
g4 = Gctx.translate g3 (2,2)
```

On the screen

Routing events
through container widgets

Event Handling: Routing

- When a container widget handles an event, it passes the event to the appropriate child
- The Gctx.gctx must be translated so that the child can interpret the event in its own local coordinates.

widget.ml

```
let border (w:widget):widget =
  { repaint = …;
    size = …;
    handle = (fun (g:Gctx.gctx) (e:Gctx.event) ->
       w.handle (Gctx.translate g (2,2)) e);
  }
```

Routing events through hpair widgets

- The event handler of an hpair must check to see whether the event should be handled by the left or right widget.
  - Check the event's coordinates against the *size* of the left widget
  - If the event is within the left widget, let it handle the event
  - Otherwise check the event's coordinates against the right child's
  - If the right child gets the event, don't forget to translate its coordinates

```
handle =
 (fun (g:Gctx.gctx) (e:Gctx.event) ->
   if event_within g e (w1.size ())
   then w1.handle g e
   else
   let g = (Gctx.translate g (fst (w1.size ()), 0)) in
     if event_within g e (w2.size ())
     then w2.handle g e
     else ())
```

19: Consider routing an event through an hpair widget constructed as shown. The event will always be propagated either to w1 or w2.

```
let hp = hpair w1 w2
```

True

0%

False

0%

Consider routing an event through an hpair widget constructed by:

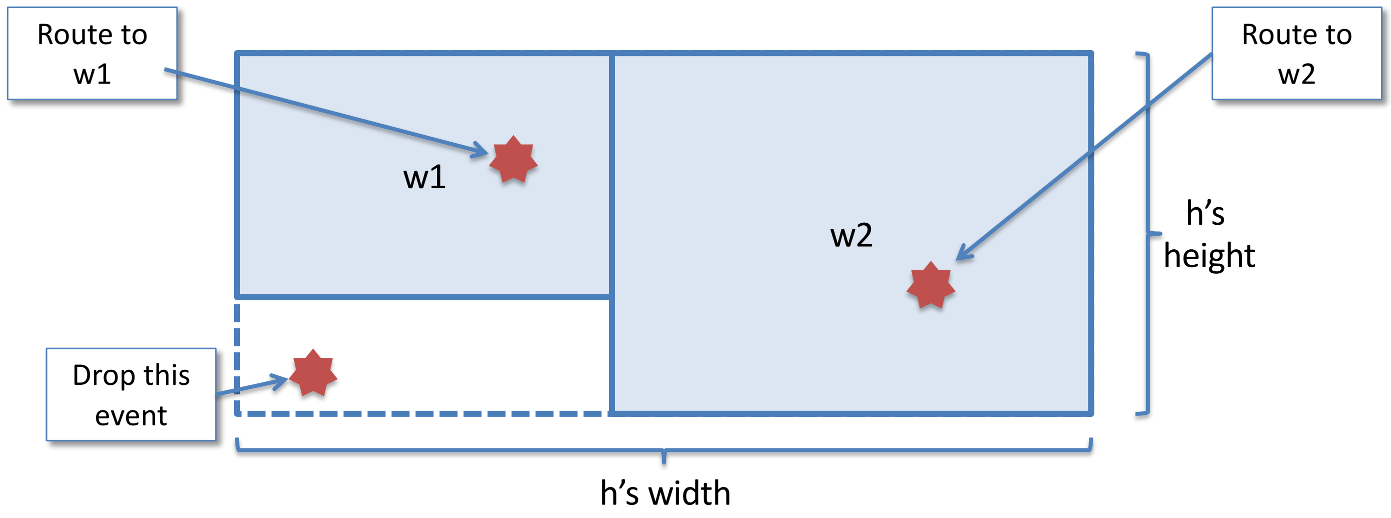
```
let hp = hpair w1 w2
```

The event will always be propagated either to w1 or w2.

1. True
2. False

Answer: False

Routing events through hpair widgets



- There are three cases for routing in an hpair.
- An event in the "empty area" should not be sent to either w1 or w2.

Stateful Widgets

How can widgets react to events?

A plain (stateless) `label` widget

```
let label (s:string) : widget =
{
  repaint = (fun (g:Gctx.gctx) -> Gctx.draw_string g (0,0) s);
  handle  = (fun _ _ -> ());
  size = (fun () -> Gctx.text_size s)
}
```

first stab at a
A stateful label Widget

```
let label (s: string) : widget =
  let r = { contents = s } in
  { repaint = (fun (g: Gctx.gctx) -> Gctx.draw_string g (0,0) r.contents);
    handle  = (fun _ _ -> ());
    size    = (fun () -> Gctx.text_size r.contents)
  }
```

- The label "constructor" creates an object: a record r containing a mutable string plus “methods” that can access this mutable string.
- *Question*: how can users **update** this string in response to an event?

  (r is "local" state -- accessible only by methods)
- *Answer*: The label constructor should give them a way to do it.

A stateful label Widget

widget.ml

```
type label_controller = { set_label: string -> unit;
                           get_label: unit -> string  }

let label (s: string) : widget * label_controller =
  let r = { contents = s } in
   ({ repaint = (fun (g: Gctx.gctx) ->
                 Gctx.draw_string g (0,0) r.contents);
      handle  = (fun _ _ -> ());
      size    = (fun () -> Gctx.text_size r.contents)
    }
   ,
   { set_label = (fun (s: string) -> r.contents <- s);
     get_label = (fun () -> r.contents);
   }
)
```

A *controller* gives access to shared state.

A `label_controller` includes two methods: accessing (getting) and updating (setting) the string.